Pointers

Danilo Ardagna

Politecnico di Milano
danilo.ardagna@polimi.it

POLITECNICO DI MILANO

Raw Pointers and Smart Pointers

- Raw Pointers:
  - You already know, C pointers with `new` and `delete` instead of low level `malloc()` and `free()`
  - Careful use, memory leak!!!
- Smart Pointers:
  - Managed by the compiler, no garbage collector
  - Allocated objects have a counter associated
  - When the counter becomes equal to 0, heap memory is automatically released
  - Nice but still careful use and technicalities, we will go back to this later during this course

Raw Pointers – Building from the ground up

direct memory addresses

- The hardware provides memory and addresses
  - Low level
  - Untyped
  - Fixed-sized chunks of memory
  - No checking
  - As fast as the hardware architects can make it
- The application builder needs something like a vector
  - Higher-level operations
  - Type checked
  - Size varies (as we get more data)
  - Close to optimally fast

The computer's memory

- As a program sees it
  - Local variables "live on the stack"
  - Global variables are "static data"
  - The executable code is in "the code section"
  - "Free store" is managed by `new` and `delete`

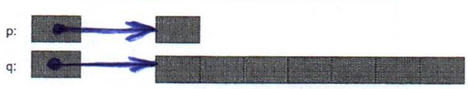
memory layout:

Code
Static Date → Global variables
Free store → Here everything is allocated with "new"
Stack → where the environments of the functions are allocated when the functions are called

The free store
(sometimes called "the heap")

- You request memory "to be allocated" "on the free store" by the `new` operator
  - **The new operator returns a pointer to the allocated memory**
  - A pointer is the address of the first byte of the memory
  - For example
    - `int* p = new int;` **// allocate one uninitialized int**
    - `int* q = new int[7];` **// allocate seven uninitialized ints // "an array of 7 ints"**
    - `double* pd = new double[n];` **// allocate n uninitialized doubles**
  - A pointer points to an object of its specified type
  - A pointer does not know how many elements it points to

it's for an array of 7



* Because of this, when we use `int* q = new int[7];` we have to worry also about the elimination at the end (by: `delete [] q;`)

!

What is the difference between:

```
int * q = new int[7];   (1)
int q[7];               (2)
```
?

(1) We're allocating dynamically into the heap an array of 7 elements

(2) We have to know the "7" at compile time (in (1) we could write: `cin>>n; int*q = new int[n];`)

Moreover:

Code	
Static data	
Free store	(1)
Stack	(2)

*

Pointer states

- The value (i.e., the address) stored in a pointer can be in one of four states:
  1. It can point to an object
  2. It can point to the location just immediately past the end of an object
  3. It can be a null pointer, indicating that it is not bound to any object
  4. It can be invalid; values other than the preceding three are invalid

!

- **It is an error to copy or try to access the value of an invalid pointer** !
  - as when we use an uninitialized variable, this error is one that the compiler is unlikely to detect
- The result of accessing an invalid pointer is **undefined** → the behaviour is random
- We must always (hopefully) know whether a given pointer is valid

Null pointers (C++ 11 style)

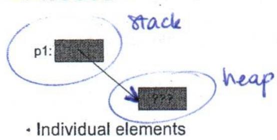
- A **null pointer** does not point to any object. Code can check whether a pointer is null before attempting to use it.

```
int *p1 = nullptr;
```

- `nullptr` is a literal that has a special type that can be converted to any other pointer type

```
int *p1 = nullptr;
// some code..
if (p1 == nullptr) {
    cout << "we cannot access";
}
else {
    *p1 = 5;
    cout << *p;
}
```

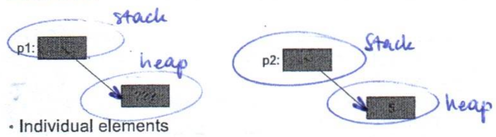

Access



- Individual elements

```
int* p1 = new int;        // get (allocate) a new uninitialized int
```


Access



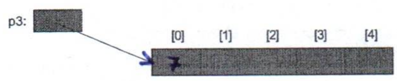
- Individual elements

```
int* p1 = new int;        // get (allocate) a new uninitialized int
int* p2 = new int(5);     // get a new int initialized to 5

int x = *p2;              // get/read the value pointed to by p2
                          // (or "get the contents of what p2 points
                          //  to")
                          // in this case, the integer 5

int y = *p1;              // undefined: y gets an undefined value;
                          // don't do that!!!!
```


Access



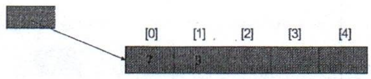
- Arrays (sequences of elements)

```
int* p3 = new int[5]; // get (allocate) 5 ints
                      // array elements are numbered [0], [1], [2],
...
p3[0] = 7;            // write to ("set") the 1st element of p3
```

! Once we do this, we can use p3 as an array!

Access

p3:



- Arrays (sequences of elements)

```
int* p3 = new int[5]; // get (allocate) 5 ints
                      // array elements are numbered [0], [1], [2],
...
p3[0] = 7;            // write to ("set") the 1st element of p3
p3[1] = 9;

int x2 = p3[1];       // get the value of the 2nd element of p3

int x3 = *p3;         // we can also use the dereference operator *
                      // for an array
                      // *p3 means p3[0] (and vice versa)
```

!!! x3 gets the value to what p3 is pointing. p3 is pointing to the first element and so:
$*p3 = p3[0]$

We can use a pointer as an array

Why use free store?

- To allocate objects that have to outlive the function that creates them:
  - For example

```
double* make(int n)          // allocate n doubles
{
    return new double[n];
}
```


Why use free store?

- With old C, when you do not know a priori your data structure size and you do not want to over-allocate memory
  - For this purpose in C++ use STL containers
- When you want to share large data structures and avoid multiple copies
  - Copies waste memory
  - Copies need to be kept in sync and this introduces additional overhead (and we may also forget!!!)

only use for raw pointers

Access

- A pointer **does not know** the number of elements that it's pointing to (only the address of the first element)

```
double* p1 = new double;
*p1 = 7.3;            // ok
p1[0] = 8.2;          // ok
p1[17] = 9.4;         // ouch! Undetected error
p1[-4] = 2.4;         // ouch! Another undetected error

double* p2 = new double[100];
*p2 = 7.3;            // ok
p2[17] = 9.4;         // ok
p2[-4] = 2.4;         // ouch! Undetected error
```

p1:

p2:

Access

- A pointer **does not know** the number of elements that it's pointing to (only the address of the first element)

```
double* p1 = new double;
*p1 = 7.3;            // ok
p1[0] = 8.2;          // ok
p1[17] = 9.4;         // ouch! Undetected error
p1[-4] = 2.4;         // ouch! Another undetected error

double* p2 = new double[100];
*p2 = 7.3;            // ok
p2[17] = 9.4;         // ok
p2[-4] = 2.4;         // ouch! Undetected error
```

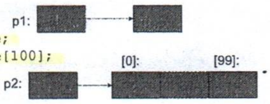
p1:

p2:

Access

- A pointer **does not know** the number of elements that it's pointing to

```
double* p1 = new double;
double* p2 = new double[100];
```



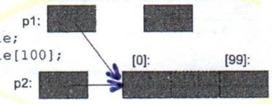

```
p1[17] = 9.4; // error (obviously)
p1 = p2;      // assign the value of p2 to p1
```

(after the assignment)

Access

- A pointer **does not know** the number of elements that it's pointing to

```
double* p1 = new double;
double* p2 = new double[100];
```




```
p1[17] = 9.4; // error (obviously)
p1 = p2;      // assign the value of p2 to p1
```

(after the assignment)

```
p1[17] = 9.4;  // now ok: p1 now points to the array of 100
               // doubles
```

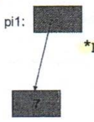

Access

- A pointer ***does*** know the type of the object that it's pointing to

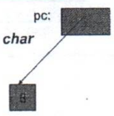
```
int* pi1 = new int(7);
int* pi2 = pi1;  // ok: pi2 points to the same object as pi1
double* pd = pi1; // error: can't assign an int* to a double*
char* pc = pi1;   // error: can't assign an int* to a char*
```

!!!

- **There are no implicit conversions between a pointer to one value type to a pointer to another value type**
- However, there are implicit conversions between value types:




```
*pc = 8;     // ok: we can assign an int to a char
```



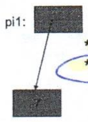
we cannot rely on automatic conversion of the pointers (between pointers) but we can transform pointers through values!

Access

- A pointer ***does*** know the type of the object that it's pointing to

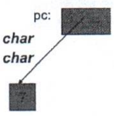
```
int* pi1 = new int(7);
int* pi2 = pi1;  // ok: pi2 points to the same object as pi1
double* pd = pi1; // error: can't assign an int* to a double*
char* pc = pi1;   // error: can't assign an int* to a char*
```

- **There are no implicit conversions between a pointer to one value type to a pointer to another value type**
- However, there are implicit conversions between value types:



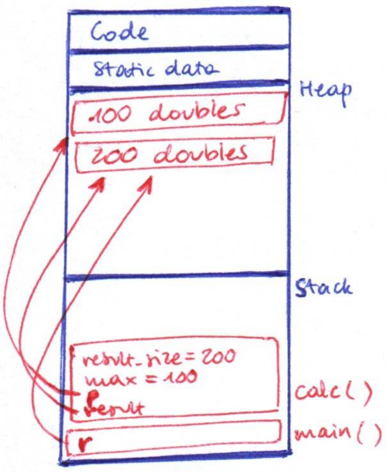

```
*pc = 8;     // ok: we can assign an int to a char
*pc = *pi1;  // ok: we can assign an int to a char
```

!



Pointers, arrays, and STL containers

- With pointers and arrays we are "touching" hardware directly with only the most minimal help from the language. Here is where serious programming errors can most easily be made, resulting in malfunctioning programs and obscure bugs
  - Be careful and operate at this level only when you really need to
  - If you get "**segmentation fault**", "**bus error**", or "**core dumped**", suspect an uninitialized or otherwise invalid pointer
- `vector` (and other STL containers) is one way of getting almost all of the flexibility and performance of arrays with greater support from the language (read: fewer bugs and less debug time)



Thanks to 'result' the 200 doubles are accessible by 'r' even outside 'calc()', however the 100 doubles created for 'p' are inaccessible out of 'calc()' and so it's a waste of memory.
How can we fix it?
delete[ ]

because we have a piece of memory allocated (it'll be allocated till the end of the program)

A problem: memory leak

1.

```
double* calc(int result_size, int max)
{
    double* p = new double[max]; // allocate another max
                                 // doubles
                                 // i.e., get max doubles from
                                 // the free store
    double* result = new double[result_size];
    // ... use p to calculate results to be put in result ...
    return result;
}

double* r = calc(200,100); // oops! We "forgot" to give the memory
                           // allocated for p back to the free
                           // store
```

Is there "new"? There MUST BE a "**delete**"

- **Lack of de-allocation** (usually called "memory leaks") can be a serious problem in real-world programs
- A program that must run for a long time can't afford any memory leaks

A problem: memory leak

```
double* calc(int result_size, int max)
{
    int* p = new double[max]; // allocate another max doubles
                              // i.e., get max doubles from the
                              // free store
    double* result = new double[result_size];

    // ... use p to calculate results to be put in result ...

    delete[ ] p;              // de-allocate (free) that array
                              // i.e., give the array back to the
                              // free store

    return result;
}

double* r = calc(200,100);
// use r
delete[ ] r;                  // easy to forget
```


Memory leaks

- A program that needs to run "forever" can't afford any memory leaks
  - An operating system is an example of a program that "runs forever"
- All memory is returned to the system at the end of the program
  - If you run using an operating system (Windows, Unix, whatever)
- Program that runs to completion with predictable memory usage may leak without causing problems
  - i.e., memory leaks aren't "good/bad" but they can be a major problem in specific circumstances

Memory leaks

- Another way to get a memory leak

2.

```
void f()
{
    double* p = new double[27];
    // ...
    p = new double[42];
    // ...
    delete[] p;
}
```

p:
1st value
2nd value

// 1st array (of 27 doubles) leaked

these are still allocated even if now they're un-reachable.
Once we lose the pointer we cannot access and we cannot release.

Memory leaks

- How do we systematically and simply avoid memory leaks?
  - don't mess directly with `new` and `delete`
    - Use vector, etc.
  - or use a garbage collector
    - A garbage collector is a program that keeps track of all the memory you allocated dynamically
    - In C++ we have Smart Pointers!
      - Allocate and return unused free-store allocated memory to the free store
    - Unfortunately, even a garbage collector and Smart Pointers do not prevent all leaks

Smart pointers associate a counter to objects and whenever we don't have a pointer pointing to an object the counter becomes zero and the object is deallocated

!!

Free store summary

- Allocate using new
  - New allocates an object on the free store, sometimes initializes it, and returns a pointer to it
    - `int* pi = new int;` // default initialization (none for int)
    - `char* pc = new char('a');` // explicit initialization
    - `double* pd = new double[10];` // allocation of (uninitialized) array
- Deallocate using delete and delete[ ]
  - delete and delete[ ] return the memory of an object allocated by new to the free store so that the free store can use it for new allocations
    - `delete pi;` // deallocate an individual object
    - `delete pc;` // deallocate an individual object
    - `delete[ ] pd;` // deallocate an array
- Delete of the null pointer does nothing
  - `char *p = nullptr;`
  - `delete p;` // harmless

References

- Lippman Chapters 2 and 3

Credits

- Bjarne Stroustrup. www.stroustrup.com/Programming